Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi

Jianbin Fang, Ana Lucia Varbanescu, Baldomero Imbernón, José M. Cecilia, and Horacio Peréz-Sánchez

Parallel and Distributed Systems Group
Delft University of Technology, Delft, the Netherlands
j.fang@tudelft.nl
Informatics Institute
University of Amsterdam, Amsterdam, the Netherlands
a.l.varbanescu@uva.nl
Bioinformatics and High Performance Computing Research Group (BIO-HPC)
Computer Science Department
Universidad Católica San Antonio de Murcia (UCAM), Guadalupe E30107, Spain
bimbernon@alu.ucam.edu
{jmcecilia,hperez}@ucam.edu

Abstract. Currently, medical research for the discovery of new drugs is increasingly using Virtual Screening (VS) methods. In these methods, the calculation of the non-bonded interactions, such as electrostatic or van der Waals, plays an important role, representing up to 80% of the total execution time. These are computationally intensive operations, and massively parallel in nature, so they perfectly fit in the new landscape of high performance computing, dominated by massively parallel architectures. Among those architectures, the latest releases by Intel and Nvidia - Xeon Phi and K20x (Kepler), respectively - are extremely interesting in terms of both performance and complexity. In this work, we discuss the effective parallelization of the non-bonded electrostatic computation for VS, and evaluate its performance on these two architectures. We empirically demonstrate that both GPUs and Intel Xeon Phi are well suited architectures for the acceleration of non-bonded interaction kernels. Further, we observe that single precision calculations for relatively small sized systems are more suitable for GPUs (K20x completely outperforms Xeon Phi), while for large systems, they achieve a similar order of magnitude performance.

Keywords: Drug Discovery, Virtual Screening, GPUs, Intel Xeon Phi, HPC

1 Introduction

The discovery of new drugs can enormously benefit from the use of Virtual Screening (VS) methods [1]. The approaches used in VS methods differ mainly

in the way they model the interacting molecules, but all of screen databases of chemical compounds containing up to millions of ligands [2].

Larger databases increase the chances of generating hits or leads, but the computational time needed for the calculations increases not only with the size of the database but also with the accuracy of the chosen VS method. Fast docking methods with atomic resolution require a few minutes per ligand [3], while more accurate molecular dynamics-based approaches still require hundreds or thousands of hours per ligand [4]. Therefore, the limitations of VS predictions are directly related to the available computational resources: the lack of such resources eventually prevents the use of detailed, high-accuracy models for VS.

In most of the VS methods, the biological system is represented in terms of interacting particles. For the calculation of the interaction energies, classical potentials are commonly used, separated into bonded and non-bonded terms. The latter describe interactions between all the elements of the system. The relevant non-bonded potentials used in VS calculations are the Coulomb and the Lennard-Jones potentials, since they accurately describe the most important short and long range interactions between protein and ligand atoms.

In VS methods the most intensive computations are spent in the calculation of non-bonded kernels. For example, in Molecular Dynamics, this computation takes up to 80% of the total execution time [5] of the application, thus becoming a computation bottleneck. It has been shown that the parallelization and optimization of the calculation of non-bonded kernels [6] permits VS methods to deal with more complex systems, simulate longer time scales or screen larger databases.

High Performance Computing (HPC) solutions [7, 8] have demonstrated they can increase considerably the performance of the different VS methods, as well as the quality and quantity of the conclusions we can get from screening. In addition, HPC platforms such as Graphics Processing Units (GPUs) [9], and more recently Intel Xeon Phi [10], provide unprecedented chip-level performance, with increased parallelism and peak performance.

GPUs have been widely applied in many different fields of applications [11, 12], and concretely in VS methods [13] [14]. Moreover, driven by the video game market, their prices and energy consumption are very low. All Nvidia GPU platforms can be programmed using the Compute Unified Device Architecture (CUDA) programming model, essentially allowing GPUs to operate and be seen as highly parallel computing devices.

As Intel's Xeon Phi is a much newer platform, there are no studies concerning its applicability for accelerating VS calculations. In this work, we show a performance evaluation for a VS method on these two emerging massively parallel architectures: Intel Xeon Phi and Nvidia Kepler-based GPUs, aiming to recommend the scenarios (if any) for which each platform is better suited.

2 Background

This Section introduces the particularities of the targeted architectures and programming models we have used to develop our VS simulations.

2.1 The CUDA Programming Model

All GPGPU platforms from Nvidia can be programmed using CUDA. In this way, the GPU becomes, to the programmer, a massively parallel processor to be used for highly parallel workloads.

Each GPU device is a scalable processor array consisting of a set of SIMT (Single Instruction Multiple Threads) multiprocessors (SM) [9], each of them containing several stream processors (SPs). Different memory spaces are also available. The global memory (also called device or video memory) is the only space accessible to all multiprocessors. It is the largest and the slowest memory available to the GPU. Moreover, each multiprocessor has its own memory space, called shared memory. The shared memory and faster, than global memory [15]. The most recent GPUs also feature two levels of caches - per SM and per chip. Finally, local memory is available per SP.

The CUDA programming model is based on a hierarchy of abstraction layers. A **thread** is the basic execution unit that is mapped to a single SP. A **block** is a batch of threads which can cooperate together because they are assigned to the same multiprocessor, and therefore they share all the resources included in this multiprocessor, including the register file and the shared memory. A **grid** is composed of several blocks which are mapped and scheduled (by the hardware) on the multiprocessors. Finally, the threads included in a block are divided into batches of 32 threads called **warps**. The warp is the scheduled unit, so the threads of the same block are scheduled in a given multiprocessor warp by warp.

The programmer declares the number of blocks, the number of threads per block and their distribution to arrange parallelism given the program constraints (i.e., data and control dependencies).

CUDA offers a new parallel programming model that combines several traditional ones. The warps are executed in a pure Single-Instruction Multiple-Data (SIMD) fashion, although serialization of threads within a warp is allowed by the model. This execution resembles somehow a vectorized execution in CMPs with vector length of 32 [16]. In a coarse-grained execution, different blocks may execute different instructions at the same time, being executed in a Multiple-Instruction Multiple-Data (MIMD) fashion. Therefore, the model provides two-levels of parallelism [17] that should be manage by the programmer. Finally, the complete execution of a kernel in the GPU is performed in a Single-Program Multiple-Data (SPMD) fashion, being even possible in the latest generation of GPUs execute several programs at the same time. In this paper, we use Nvidia Tesla K20x and CUDA version 5.5.

2.2 Intel Xeon Phi Programming

Intel Xeon Phi has over 50 cores (the version used in this paper belongs to the 5100 series and has 60 cores) connected by a high-performance on-die bidirectional interconnect (shown in Figure 1). In addition to these cores, there are 16 memory channels (supported by memory controllers) delivering up to 5.0 GT/s [18]. When working as an accelerator, Phi can be connected to a host (i.e., a device that manages it) through a PCI Express (PCIe) system interface - similar to GPU-like accelerators. Different from GPUs, a dedicated embedded Linux μOS (version: `2.6.38.8`) runs on the platform.

Each core contains a 512-bit wide vector unit (VPU) with vector register files (32 registers per thread context). Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a core-private 512KB unified L2 cache. In total, a 60-core machine has a total of 30MB of L2 cache on the die. The L2 caches are kept fully coherent by the hardware, using DTDs (distributed tag directories), which are referenced after an L2 cache miss. Note that the tag directory is not centralized, but split up into 64 DTDs, each getting an equal portion of the address space and being responsible for maintaining it globally coherent.

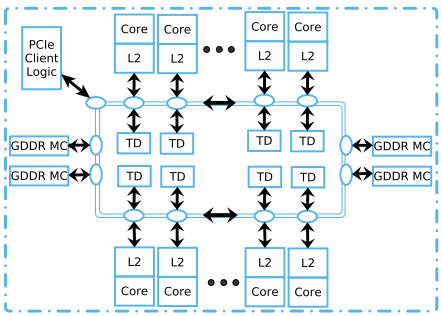


Fig. 1. The Intel Xeon Phi Architecture.

In terms of usability, there are two ways an application can use Intel Xeon Phi: (1) in *offload mode* - the main application is running on the host, and it only offloads selected (highly parallel, computationally intensive) work to the coprocessor, or (2) in *native mode* - the application runs independently, on the Xeon Phi only, and can communicate with the main processor or other coprocessors [19] through the system bus. In this work, we use Xeon Phi in the *native* mode. Being an `x86` SMP-on-a-chip architecture, Xeon Phi offers the full capability to use the same tools, programming languages, and programming models as a regular Intel Xeon processor. Specifically, tools like Pthreads, OpenMP, Intel Cilk Plus, and OpenCL are readily available. Given the large number of cores on the platform, a dedicated MPI version is also available. In this paper,

Table 1. A comparison of the selected processors.

	Tesla K20x	Phi 5110P
Cores	2688	60
Core Clock (MHz)	732	1053
SP TFLOPS	3.95	2.02
DP TFLOPS	1.31	1.01
Memory Size (GB)	6	8
Mem. Bandwidth (GB/s)	250	320
Price (USD)	3500	2650
Max Power Usage	235	225

we select the native programming model (i.e., OpenMP) for Xeon Phi, and use Intel's `icc` compiler (`V13.1.1.163`) for Xeon Phi.

2.3 A Comparison of K20x and Phi

We compare the 'officially' released numbers of Nvidia K20x and Xeon Phi 5110P in Table 1 [1] [2]. We note that Tesla K20x run 2× as fast as Xeon Phi in SP Flops while their performance is similar in DP Flops. Furthermore, they consume similar amount of power.

3 Methodology

3.1 Sequential Baseline

Algorithm 1 The sequential pseudocode.

```
1: for i = 0 to nrec do
2:     for j = 0 to nlig do
3:         calculus(rec[i], lig[j])
4:     end for
5: end for
```

In our study we focus on the particular case of protein-ligand docking, and concretely, in the calculation of the electrostatic potential kernel show in Algorithm 1. This is the baseline for several methodologies used in VS methods, such as Molecular Dynamics and protein-protein docking. Both the receptor and ligand molecules are represented by *rec* and *lig* particles, which are specified by their positions and charges. The system has *nrec* and *nlig* atoms of *rec* and *lig*, respectively. The computation inside the double loop calculates the electrostatic

[1] Nvidia Tesla K20x: `http://www.nvidia.com/object/tesla-servers.html`
[2] Intel Xeon Phi 5110P: `http://ark.intel.com/products/71992/`

potential and is expressed as $q[i]q[j]/r_{ij}$ where $q[i]$ and $q[j]$ are related to charges for individual particles for receptor and ligand, and r_{ij} is the distance between respective receptor and ligand atoms.

3.2 Implementation on the GPU

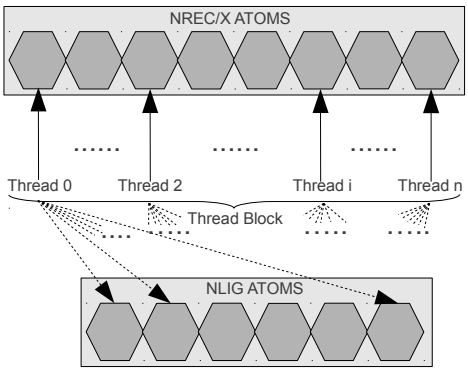


Fig. 2. CUDA design for X thread blocks (with $X = 1$) with n threads layout.

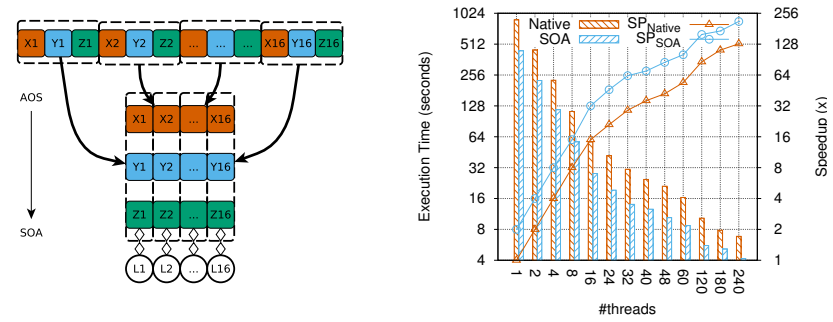
Our starting point is a CUDA implementation previously presented in [14]. Figure 2 shows this design. Each atom from the receptor molecule is represented by a single thread. Then, every CUDA thread goes through all the atoms of the ligand molecule. The double parallelism within CUDA is exploited by

1. each thread performing the energy calculations with the entire ligand data.
2. having as many threads as *nrec* atoms.
3. having as many thread blocks as the number of *nrec* atoms divided by the number of threads within a block. (a configuration parameter of our application).

We also enable a tiling technique to take advantage of the data locality, and thus increasing the memory bandwidth of our application. Specifically, we group atoms of the ligand molecule in tiles, facilitating threads to collaborate in order to bring that information to the *shared memory*. More details on this tecnique can be found in [14].

3.3 Intel Xeon Phi Implementation

As mentioned before, our choice for programming the Xeon Phi is OpenMP. OpenMP is a pragma-based, high level programming model. Typically, programmers need to identify the potential hotspots (typically, loops) in a compute-intensive kernel and annotate them with specific parallelization pragmas. In turn, these regions will be parallelized with the help of the compiler.



(a) (b)

Fig. 3. Vectorization on Xeon Phi: (a) The AOS-to-SOA Scheme for vectorization ((X, Y, Z) represents the coordination of an atom; L represents SIMD lanes and Xeon Phi has 16 lanes when using single-precision data elements), (b) The performance (the execution time in seconds and the speedup over the native AOS format) of the transformed data structures (SOA). Note the Y axis is in log scale.

A straightforward way to parallelize our kernel using OpenMP is to add an `omp parallel` construct over the outer loop (Line 1) of Algorithm 1. Further, we map the inner loop onto the vector core (512-bits), where the distance r_{ij} between atom i and atom j is calculated [3]. This first version is, however, naive in terms of implementation and poor in terms of performance.

Next, we took a closer look at the `calculus` in Algorithm 1, and found that the kernel uses data structures in the form of `Array of Structure (AOS)`. Specifically, the coordinates (X, Y, Z) of each atom are stored contiguously (shown in Figure 3(a)). Consequently, a vector thread first loads 16 X values $(X_1, X_2, ..., X16)$ and then 16 Y values $(Y_1, Y_2, ..., Y16)$, and then 16 Z values $(Z_1, Z_2, ..., Z16)$. In other words, it needs to gather non-contiguous data from memory (i.e., 16 single-precision floating-point data elements located far from each other in memory). The required data elements may even fall into separate cache-lines. This non-contiguous pattern can lead to inefficient cache/bandwidth usage, and therefore limits the performance of vectorization. To tackle this issue, we transformed the data structures into the more parallelism-friendlier `Structure of Array (SOA)` by storing the data elements of the same dimension contiguously (Figure 3(a)). Consequently, the same operations of calculating the distance between atoms are performed on data elements within a single cache-line.

The execution time (when $nrec = 10240000$ and $nlig = 8192$) of the native kernel and the transformed kernel is shown in Figure 3(b). When using 240 threads, the native version performs the best (6848 ms) and we note that the

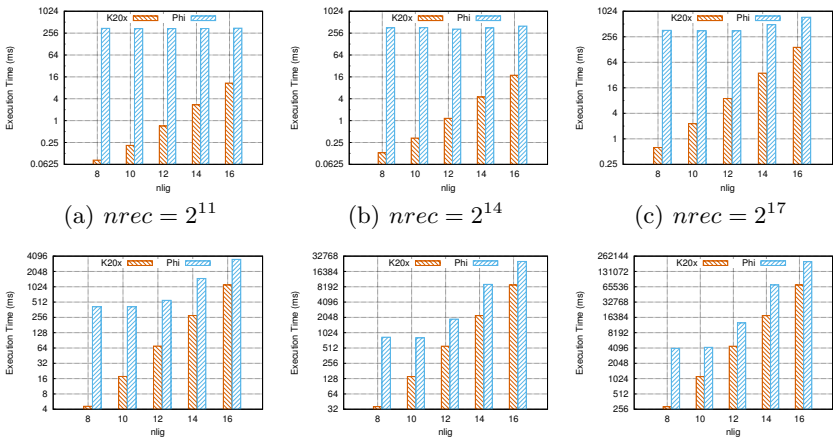
[3] When enabling the auto-vectorization module (using the -O2/-O3 option), the inner loop can be successfully vectorized by the compiler.

execution decreases linearly with the number of threads, indicating a good scalability on the VS application. After using the SOA data structure on Phi, the performance has double - the code runs 2× faster (on average) than the first version: 4159 ms when using 240 threads.

4 Results and Discussion

Figure 4 presents a comparison of the performance of our application on the NVIDIA K20x and the Intel Xeon Phi. Note that we use the *compact* thread affinity with 240 threads on Xeon Phi, and we record the kernel execution time for both K20x and Xeon Phi. Overall, we see that K20x performs better than Xeon Phi.

When the data sets are small, the performance gap is very large. This is because the code running on Xeon Phi includes the parallelization overhead, which can be quite large when launching over 200 threads - e.g., it takes around 350 ms when $nrec \leq 17$. On the other hand, for larger data sets (e.g., $nrec = 2^{26}$ and $nlig = 2^{16}$), the code on Phi runs only 3× slower than that on K20x. As shown in Table 1, K20x has a much higher peak SP performance (4 TFlops, 2× more than the Phi), which explains a part of the observed slowdown. The remaining 1.5× optimization space on Xeon Phi is left for further exploration. Further, we expect that the performance when using DP computation (i.e., on double-precision data elements) will be much closer.



(a) $nrec = 2^{11}$ (b) $nrec = 2^{14}$ (c) $nrec = 2^{17}$

(d) $nrec = 2^{20}$ (e) $nrec = 2^{23}$ (f) $nrec = 2^{26}$

Fig. 4. Performance comparison between K20x and Xeon Phi. Note that the Y axis is in log scale.

5 Conclusions and Outlook

To summarize, our experience with Virtual Screening on NVIDIA K20x and Intel Xeon Phi has lead us to the following observations:

- Porting legacy (sequential) code in OpenMP for Xeon Phi comes almost for free. However, optimizing the outcome is relatively time-consuming, as a thorough understanding of the architectural features of the processor is mandatory.
- On Xeon Phi, it is essential to select suitable data structures (SOA instead of AOS, for caching) to enable the full utilization of the SIMD units. By comparison, GPUs like Nvidia K20x prefer the AOS-style data structures.
- Nvidia K20x significantly outperforms Intel Xeon Phi on Virtual Screening when using single-precision floating-point data elements. We expect the performance for double precision computations to be much closer.
- Comparing different implementations in native programming models is excellent for providing performance numbers for the end-users, but provides limited insight for a head-to-head comparison of the two platforms. A common solution, in a common programming model (e.g., OpenCL or OpenACC) is necessary for such a detailed analysis.

For further studies, we plan to evaluate the double precision computation for both the GPU and the Xeon Phi. Further, aiming to use a unified programming model, we will evaluate an OpenCL solution for VS on both GPUs and Xeon Phi, thus evaluating the impacts of the chosen programming model [20] on the overall performance of the application.

Acknowledgements

This work was partially supported by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA−CIEMAT), funded by the European Regional Development Fund (ERDF). CETA−CIEMAT belongs to CIEMAT and the Government of Spain. The authors also thankfully acknowledge the computer resources and the technical support provided by the Plataforma Andaluza de Bioinformática of the University of Málaga. We also thank NVIDIA for hardware donation under CUDA Teaching Center 2014. This work was partially funded by CSC (China Scholarship Council), and the National Natural Science Foundation of China under Grant No.61103014 and No.11272352.

References

1. W. L. Jorgensen, "The Many Roles of Computation in Drug Discovery," *Science*, vol. 303, pp. 1813–1818, 2004.
2. J. J. Irwin and B. K. Shoichet, "ZINC–a free database of commercially available compounds for virtual screening," *Journal of Chemical Information and Modeling*, vol. 45, no. 1, pp. 177–182, 2005.

3. Z. Zhou, A. K. Felts, R. A. Friesner, and R. M. Levy, "Comparative performance of several flexible docking programs and scoring functions: enrichment studies for a diverse set of pharmaceutically relevant targets," *Journal of Chemical Information and Modeling*, vol. 47, no. 4, pp. 1599–1608, 2007.
4. J. Wang, Y. Deng, and B. Roux, "Absolute Binding Free Energy Calculations Using Molecular Dynamics Simulations with Restraining Potentials," *Biophys. J.*, vol. 91, no. 8, pp. 2798–2814, Oct. 2006.
5. S. K. Kuntz, R. C. Murphy, M. T. Niemier, J. Izaguirre, and P. M. Kogge, "Petaflop computing for protein folding," in *In Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 12–14.
6. H. Pérez-Sánchez and W. Wenzel, "Optimization methods for virtual screening on novel computational architectures," *Curr Comput Aided Drug Des*, vol. 7, no. 1, pp. 44–52, 2011.
7. K. Kadau, T. C. Germann, and P. S. Lomdahl, "Molecular Dynamics Comes of Age: 320 Billion Atom Simulation on BlueGene/L," *International Journal of Modern Physics C*, vol. 17, no. 12, pp. 1755–1761, 2006.
8. N. D. Prakhov, A. L. Chernorudskiy, and M. R. Gaiin, "VSDocker: a tool for parallel high-throughput virtual screening using AutoDock on Windows-based computer clusters," *Bioinformatics*, vol. 26, no. 10, pp. 1374–1375, 2010.
9. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Ieee Micro*, vol. 28, no. 2, pp. 39–55, 2008.
10. Intel, "Intel Xeon Phi Coprocessor," http://software.intel.com/en-us/mic-developer, April 2013.
11. M. Morgan Kaufmann, Boston, *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*, 1st ed. Morgan Kaufmann, Feb. 2011.
12. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, pp. 13–27, Jul. 2008.
13. H. Pérez-Sánchez and W. Wenzel, "Implementation of an effective non-bonded interactions kernel for biomolecular simulations on the Cell processor," in *GI Jahrestagung*, 2009, pp. 721–729.
14. G. Guerrero, H. Pérez-Sánchez, W. Wenzel, J. Cecilia, and J. García, "Effective Parallelization of Non-bonded Interactions Kernel for Virtual Screening on GPUs," in *5th International Conference on Practical Applications of Computational Biology and Bioinformatics (PACBB 2011)*, ser. Advances in Intelligent and Soft Computing. Springer Berlin / Heidelberg, 2011, vol. 93, pp. 63–69.
15. NVIDIA Corporation, *NVIDIA CUDA C Programming Guide 4.0*, 2011.
16. V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, ser. SC 08. IEEE Computer Society, 2008, pp. 1:1–1:11.
17. J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez, "Simulation of P Systems with Active Membranes on CUDA," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 313–322, 2010.
18. Intel, *Intel Xeon Phi Coprocessor System Software Development Guide*, Nov. 2012.
19. ——, *An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors*, Oct. 2012.
20. J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *2011 International Conference on Parallel Processing (ICPP'11)*. IEEE, Sep. 2011, pp. 216–225. [Online]. Available: http://dx.doi.org/10.1109/icpp.2011.45